Proof Support for IMP++ in HOL-OCL

Master Thesis

Lukas Brügger

Information Security
ETH Zürich

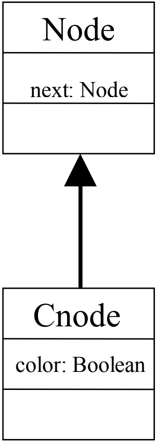
August 2, 2007

Content

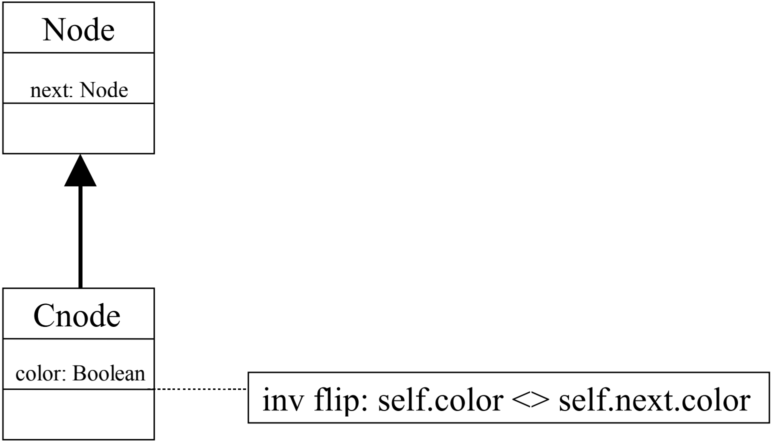
Overview

- There is a gap between verification on the specification level and on the implementation level.
- We close this gap by extending HOL-OCL ...
- ...with a programming language semantics based on a Hoare-calculus.
- All encodings are strongly typed

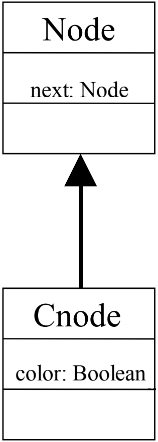
Motivation



Motivation


Node
next: Node
Cnode
color: Boolean
inv flip: self.color <> self.next.color

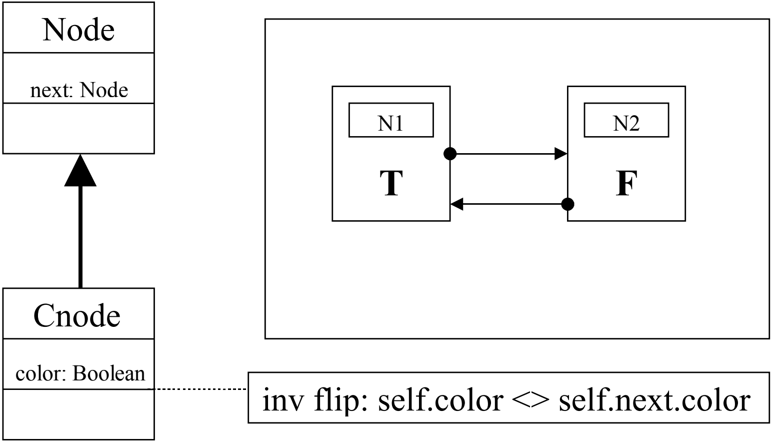
Motivation




```
N1 := new Cnode();
N2 := new Cnode();
N1.set_color true;
N2.set_color false;
N1.set_next N2;
N2.set_next N1;
return N1;
```

inv flip: self.color <> self.next.color

Motivation


Node
next: Node
Cnode
color: Boolean
N1
T
N2
F
inv flip: self.color <> self.next.color

Content

Background: HOL-OCL

HOL-OCL is an interactive proof environment for UML/OCL.

It provides:

- A datatype package for OO data structures
- A machine-checked semantics for OCL
- A proof calculi for a three-valued logic over path expressions
- A framework for analyzing OO specifications

Type constructor τ *up*: assigns to each type τ a type lifted by $\perp$.

Background: HOL-OCL

HOL-OCL is an interactive proof environment for UML/OCL.

It provides:

- A datatype package for OO data structures
- A machine-checked semantics for OCL
- A proof calculi for a three-valued logic over path expressions
- A framework for analyzing OO specifications

Type constructor τ *up*: assigns to each type τ a type lifted by $\perp$.

Missing: Programming language semantics. Therefore no program verification.

Background: Hoare Logic

Method for analysing programs with a small-step semantics. Main construct is a Hoare Triple:

$$\models \{P\}\ c\ \{Q\}$$

"if P holds for some state s and c terminates and reaches state t, then Q must hold for t"

- P,Q are assertions. Modelled as set of states
- Denotational Semantics used for relating states and commands
- Isabelle provides IMP: a Hoare calculus for an imperative language

Content

IMP++

Motivation: Extend IMP by core features of object-orientation (Java subset)

Deep embedding: Syntax is introduced as datatype definition:

```
datatype
  α com =
      SKIP
    | Cmd α cmd
    | Semi α com α com                 (_ ; _)
    | Cond α bexp α com α com          ( IF _ THEN _ ELSE _)
    | While α bexp α com               ( WHILE _ DO _)

types
   α bexp = α state => bool up
   α cmd  = α state => α state up
```

Hoare Logic for IMP++

Fairly standard, but with support for undefinedness.

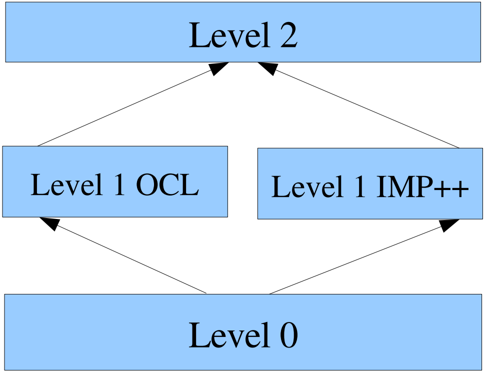
States can always be undefined. Constant *err* denotes the error state. Used for modelling exceptions.

types α assn = α state up => bool

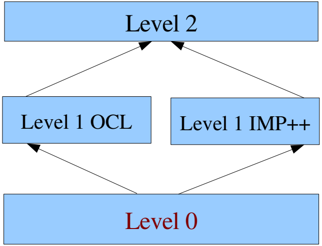
constdefs
hoare_valid :: [α assn, α com, α assn] => bool
$\models$\{P\}c\{Q\} $\equiv \forall$ s t. (s,t) $\in$ C(c) --> P s --> Q t

Content

The HOL-OCL/IMP++ Architecture



The HOL-OCL/IMP++ Architecture

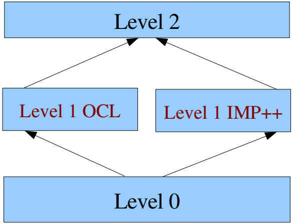


Object Store

- Universe and class types
- Getters for the attributes

Can be reused for IMP++, extended with the Setters.

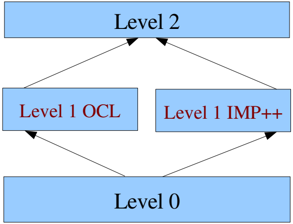
The HOL-OCL/IMP++ Architecture



Level 1:

- Lifting over the context of any semantic function
- Explicit dealing with definedness and strictness
- HOL-OCL provides types and operators to automate this

The HOL-OCL/IMP++ Architecture

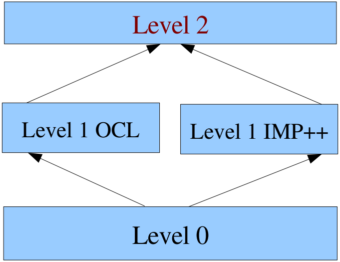


Level 1:

- The context of HOL-OCL is a pair of state (pre/post)
- The context of IMP++ is one state

⟶ Definitions and lemmas can't be reused

The HOL-OCL/IMP++ Architecture



Level 2 adds support for preconditions, postconditions and invariants.

Here we can relate the Hoare verification with the specification.

State

The commands of our language are state transitions.
The state is a mapping from oids to the Universe with the following constraints:

- NULL must not be a valid reference
- All elements in the range of the state must be defined
- There is a one-to-one correspondence between objects and their oid (oid stored as part of the object)

State

The state is a mapping from oids to the Universe with the following constraints:

```
constdefs correct_state :: "(oid ⇀ ('α U)) ⇒ bool"
" correct_state σ ≡
  (NULL ∉dom σ∧
  (∀ obj ∈ ran σ.( level0 . oclLib . is_OclAny_univ obj  ⟶
          DEF(level0. oclLib . get_OclAny obj)) ∧
  (∀ oid ∈ dom σ. level0 . oclLib . is_OclAny_univ (the (σ oid)) ⟶
          OclOidOf0(level0. oclLib . get_OclAny (the (σ oid)))=oid))"
```

The state is defined as a type definition using this invariant.

Operators: access, create, update

Large library of lemmas for the state.

SetColor

type: $Cnode => bool => state => state\ up$

SetColor

type: *Cnode => bool => state => state up*

```
consts l1_Cnode_set_color  ::
"(('a, 'b) state, 'a Cnode) VAL
 ⇒ (('a, 'b) state, bool up) VAL
 ⇒ ('a, 'b) state
 ⇒ ('a, 'b) state up"
```

SetColor

```
defs l1_Cnode_set_Color_def :
  " l1_Cnode_set_color  self  up_c σ ≡
    ( let  oid  = (l1_OclOid  self ) σ in
      let  c0  = up_c σ in
      let  self0  = access_Cnode_in_state  σ oid  in
( if  Cnode_in_state  σ oid  then
⌞(updext_Cnode_in_state  oid  (base ( Setcolor  self0  c0)) σ )⌟
else  ⊥ ))"
```

The commands of the language

Level 1 operators are:

- Setters
- update_news

They all fit in the Cmd slot provided by IMP++.

Instead `l1_Cnode_set_color N1 T` σ we write `N1 .color := T`, using syntax translation.

StackObject used for storing local variables.

The Program in IMP++

```
generate_cyclic_List  so ≡
        so .n1 :=  New(Cnode) ;
        so .n2 := New(Cnode);
        so .cn1 . color := T;
        (so .cn2) . color := F;
        (so .n1) .next  :=  (so .n2);
        (so .n2) .next  :=  (so .n1);
        so . return := (so .n1)"
```

The Calculus

Large library of lemmas about what happens during a state transition:

- Definedness of expressions and objects

The Calculus

Large library of lemmas about what happens during a state transition:

- Definedness of expressions and objects

lemma DEF_set_color:
"⟦Cnode_in_state s (I1_OclOid self s)⟧ ⟹
DEF (I1_Cnode_set_color self foo s)"

The Calculus

Large library of lemmas about what happens during a state transition:

- Definedness of expressions and objects
- Correct objects remain correct during an update

The Calculus

Large library of lemmas about what happens during a state transition:

- Definedness of expressions and objects
- Correct objects remain correct during an update

lemma Cnode_in_state_set_color :
"⟦Cnode_in_state s oid ; I1_OclOid self s = oid⟧ $\Longrightarrow$
Cnode_in_state ⌜I1_Cnode_set_color self foo s⌝ oid"

The Calculus

Large library of lemmas about what happens during a state transition:

- Definedness of expressions and objects
- Correct objects remain correct during an update
- The value of a freshly set attribute

The Calculus

Large library of lemmas about what happens during a state transition:

- Definedness of expressions and objects
- Correct objects remain correct during an update
- The value of a freshly set attribute
- The value of attributes which didn't get updated

The Calculus

Large library of lemmas about what happens during a state transition:

- Definedness of expressions and objects
- Correct objects remain correct during an update
- The value of a freshly set attribute
- The value of attributes which didn't get updated
- Objects which won't get updated remain the same

The Calculus

Large library of lemmas about what happens during a state transition:

- Definedness of expressions and objects
- Correct objects remain correct during an update
- The value of a freshly set attribute
- The value of attributes which didn't get updated
- Objects which won't get updated remain the same
- The free memory

The Calculus

Large library of lemmas about what happens during a state transition:

- Definedness of expressions and objects
- Correct objects remain correct during an update
- The value of a freshly set attribute
- The value of attributes which didn't get updated
- Objects which won't get updated remain the same
- The free memory
- Casting between class types

The Calculus

Large library of lemmas about what happens during a state transition:

- Definedness of expressions and objects
- Correct objects remain correct during an update
- The value of a freshly set attribute
- The value of attributes which didn't get updated
- Objects which won't get updated remain the same
- The free memory
- Casting between class types

The library is developped in a modular way. They are/will be created automatically by the encoder.

The Correctness Proof

$$
\begin{array}{l}
\models\{\lambda\ \sigma.\ \neg(\sigma\ \models \mathrm{err})\ \wedge \mathrm{enough\_space\_for}\ \ulcorner\sigma\urcorner 2\ \wedge \\
\quad \mathrm{is\_handle\_for\_StackObject}\ \ \mathrm{so}\ \ \mathrm{so\_oid}\ \wedge \\
\quad \mathrm{StackObject\_in\_state}\ \ulcorner\sigma\urcorner\ \mathrm{so\_oid}\} \\
\qquad\qquad \mathrm{so}\ \ .\mathrm{n1} := \ \mathrm{New}(\mathrm{Cnode}); \\
\qquad\qquad \mathrm{so}\ \ .\mathrm{n2} := \mathrm{New}(\mathrm{Cnode}); \\
\qquad\qquad \mathrm{so}\ \ .\mathrm{cn1}\ \ .\ \mathrm{color}\ \ := \mathrm{T}; \\
\qquad\qquad (\mathrm{so}\ \ .\mathrm{cn2})\ \ .\ \mathrm{color}\ \ := \mathrm{F}; \\
\qquad\qquad (\mathrm{so}\ \ .\mathrm{n1})\ \ .\mathbf{next}\ \ := \ \ (\mathrm{so}\ \ .\mathrm{n2}); \\
\qquad\qquad (\mathrm{so}\ \ .\mathrm{n2})\ \ .\mathbf{next}\ \ := \ \ (\mathrm{so}\ \ .\mathrm{n1}); \\
\qquad\qquad \mathrm{so}\ \ .\ \mathrm{return}\ \ := (\mathrm{so}\ \ .\mathrm{n1}) \\
\{\lambda\ \sigma.\ \neg(\sigma\ \models \mathrm{err})\ \wedge (\mathrm{Cnode\_inv}\ \ (\mathrm{so}\ \ .\ \mathrm{return})\ \ulcorner\sigma\urcorner)\}\text{''}
\end{array}
$$

The Correctness Proof

$\models\{\lambda\ \sigma.\ \neg(\sigma\ \models$err$)\ \wedge$ enough_space_for $\ulcorner\sigma\urcorner 2\ \wedge$
is_handle_for_StackObject so so_oid $\wedge$
StackObject_in_state $\ulcorner\sigma\urcorner$ so_oid$\}$
generate_cyclic_list so
$\{\lambda\ \sigma.\ \neg(\sigma\ \models$err$)\ \wedge$ (Cnode_inv (so .return) $\ulcorner\sigma\urcorner)\}$"

The Correctness Proof

$\models\{\lambda\ \sigma.\ \neg(\sigma \models \text{err})\ \wedge$ enough_space_for $\ulcorner\sigma\urcorner 2\ \wedge$
is_handle_for_StackObject so so_oid $\wedge$
StackObject_in_state $\ulcorner\sigma\urcorner$ so_oid $\}$
generate_cyclic_list so
$\{\lambda\ \sigma.\ \neg(\sigma \models \text{err})\ \wedge$ (Cnode_inv (so .return) $\ulcorner\sigma\urcorner$)}"

- If we start in state which is not the error state, where there's enough memory for two more objects, and where we have a StackObject of correct type, then

The Correctness Proof

$$\models \{\lambda\, \sigma.\ \neg(\sigma \models \mathrm{err}) \wedge \mathrm{enough\_space\_for}\ \ulcorner\sigma\urcorner 2 \wedge$$
$$\mathrm{is\_handle\_for\_StackObject}\ \ \mathrm{so}\ \ \mathrm{so\_oid}\ \wedge$$
$$\mathrm{StackObject\_in\_state}\ \ulcorner\sigma\urcorner\ \mathrm{so\_oid}\}$$
$$\mathrm{generate\_cyclic\_list}\quad \mathrm{so}$$
$$\{\lambda\, \sigma.\ \neg(\sigma \models \mathrm{err}) \wedge (\mathrm{Cnode\_inv}\ \ (\mathrm{so}\ \ .\mathrm{return})\ \ulcorner\sigma\urcorner)\}"$$

- If we start in state which is not the error state, where there's enough memory for two more objects, and where we have a StackObject of correct type, then
- after executing the method generate_cyclic_list on the StackObject,

The Correctness Proof

$$\models\{\lambda\ \sigma.\ \neg(\sigma\ \models \text{err})\ \wedge \text{enough\_space\_for}\ \ulcorner\sigma\urcorner 2\ \wedge$$
$$\text{is\_handle\_for\_StackObject}\ \ \text{so}\ \ \text{so\_oid}\ \wedge$$
$$\text{StackObject\_in\_state}\ \ulcorner\sigma\urcorner\ \text{so\_oid}\}$$
$$\text{generate\_cyclic\_list}\quad \text{so}$$
$$\{\lambda\ \sigma.\ \neg(\sigma\ \models \text{err})\ \wedge (\text{Cnode\_inv}\ \ (\text{so}\ \ .\text{return})\ \ulcorner\sigma\urcorner)\}\text{''}$$

- If we start in state which is not the error state, where there's enough memory for two more objects, and where we have a StackObject of correct type, then
- after executing the method generate_cyclic_list on the StackObject,
- we end in a state which is not the error state and where the returned object satisfies the `flip` invariant.

Proof Outline

- A sequence of applications of rule semi_hoare, with explicit instantiations

$$\frac{\{A\}c\{B\}; \quad \{B\}d\{C\}}{\{A\}c;d\{C\}}$$

Proof Outline

- A sequence of applications of rule semi_hoare, with explicit instantiations
- Most lemmas of the library can safely be added to the simplifier, which proves most subgoals
- Only few direct rule applications necessary
- Final step a little more difficult. Through application of weak coinduction to several definedness expression and two inequalities.

Content

Conclusion

- We showed the feasability of a typed verification approach for object-oriented programs
- Large library of lemmas and definitions provides relatively efficient calculus for a Hoare logic
- Definition and lemmas are / will be created automatically by the encoder
- Tight integration with HOL-OCL allows taking advantage of the large library, and ...
- ...provides an integrated reasoning over object-oriented specifications and programs

Future Work

- Extend the encoder
- Support for method calls
- Verification Condition Generator